Computer Science 3 - 2010

Programming Language Translation

Practical for Week 21, beginning 13 September 2010

Hand in your solutions to this practical *before* lunch time on your next practical day, correctly packaged in a transparent folder with your cover sheets. **Unpackaged and late submissions will not be accepted - you have been warned.** Please do NOT come to a practical and spend the first hour printing or completing solutions from the previous week's exercises. Since the practical will have been done on a group basis, please hand in one copy of the cover sheet for each member of the group. These will be returned to you in due course, signed by the marker. **Please make it clear whose folder you have used for the electronic submission, for example g03A1234.** Lastly, please resist the temptation to carve up the practical, with each group member only doing one task. The group experience is best when you discuss each task together.

Objectives:

In this practical you are to

- familiarize you with simple applications of the Coco/R parser generator, and
- write grammars that describe simple language features.

You will need this prac sheet and your text book. As usual, copies of the prac sheet are also available at `http://www.cs.ru.ac.za/CSc301/Translators/trans.htm`.

Outcomes:

When you have completed this practical you should understand

- how to develop context-free grammars for describing the syntax of various languages and language features;
- the form of a Cocol description;
- how to check a grammar with Coco/R and how to compile simple parsers generated from a formal grammar description.

To hand in:

This week you are required to hand in, besides the cover sheet:

- Listings of your solutions to the grammar problems, produced on the laser printer by using the LPRINT utility or UltraEdit in a small Courier font. Some of these listings will get quite "wide" so please set them out nicely.
- Electronic copies of your grammar files (ATG files).

I do NOT require listings of any Java code produced by Coco/R.

Keep the prac sheet and your solutions until the end of the semester. Check carefully that your mark has been entered into the Departmental Records.

You are referred to the rules for practical submission which are clearly stated in our Departmental Handbook. However, for this course pracs must be posted in the "hand-in" box outside the laboratory and not given to demonstrators.

A rule not stated there, but which should be obvious, is that you are not allowed to hand in another group's or student's work as your own. Attempts to do this will result in (at best) a mark of zero and (at worst) severe disciplinary action and the loss of your DP. You are allowed - even encouraged - to work and study with other students, but if you do this you are asked to acknowledge that you have done so. You are expected to be familiar with the University Policy on Plagiarism, which you can consult at:

http://www.scifac.ru.ac.za/plagiarism_policy.pdf

Task 1 - creating a working directory and unpacking the prac kit

There are several files that you need, zipped up this week in the file `PRAC21.ZIP` (Java version) or `PRAC21C.ZIP` (C# version)

- Immediately after logging on, get to the DOS command line level by using the `Start -> Command prompt` option from the tool bar.

- Copy the prac kit into a newly created directory/folder in your file space

```
j:
md  prac21
cd  prac21
copy  i:\csc301\trans\prac21.zip
unzip  prac21.zip
```

  You will find the executable version of Coco/R and batch files for running it, frame files, and various sample programs and grammars, contained in files with extensions like

```
*.ATG,    *.PAV,    *.TXT    *.BAD
```

- UltraEdit is probably your editor of choice. The version in the lab is configured to run various of the compilers easily, and has been tweaked to run Coco/R in much the same sort of way (see below). *To get this to work properly, start UltraEdit from a command window by giving the command* `UEDIT32`*, rather than by clicking on an icon on the desktop.*

Task 2 - Simple use of Coco/R - a quick task

In the kit you will find `Calc.atg`. This is essentially the calculator grammar on page 106 of the text, with a slight (cosmetic) change of name.

Use Coco/R to generate a parser for data for this calculator. You do this most simply by giving the command

```
cmake  Calc
```

The primary name of the file (`Calc`) is case sensitive. Note that the `.ATG` extension is needed, but not given in the command. Used like this, Coco/R will simply generate three important components of a calculator program - the parser, scanner, and main driver program. Cocol specifications can be programmed to generate a complete calculator too (ie one that will evaluate the expressions, rather than simply check them for syntactic correctness), but that will have to wait for the early hours of another day.

(Wow! Have you ever written a program so fast in your life before?)

Of course, having Coco/R write you a program is one thing. But it might also be fun and interesting to run the program and see that it works.

A command like

```
crun  Calc  calc.txt
```

will run the program `Calc` and try to parse the file `calc.txt`, sending error messages to the screen. Giving the command in the form

```
crun  Calc  calc.bad  -L
```

will send an error listing to the file `listing.txt`, which might be more convenient. Try this out.

Well, you did all that. Well done. What next?

For some light relief and interest you might like to look at the code the system generated for you (three `.java` files are created in a subdirectory `Calc`) - you don't have to comment this week, simply gaze in awe. Don't take too long over this, because now you have the chance to be more creative.

That's right - we have not finished yet. Modify the grammar so that you can use parentheses in your expressions, allow leading unary + or - signs, can raise quantities to a "power" (as in `12↑2 + 5↑6`, meaning $12^2 + 5^6$) and also give the calculator a square-root capability (as in `4 + sqrt(5 * 12))`. Oh, and while you are at it. allow it to recognize numbers that incorporate a decimal point, as in `3.4` or `3.` or `.45`

Of course, the application does not have any real "calculator" capability - it cannot calculate anything (yet). It only has the ability to recognise or reject expressions at this stage. Try it out with some expressions that use the new features, and some that use them incorrectly.

Warning. Language design and grammar design is easy to get wrong. Think hard about these problems before you begin, and while you are doing them.

Task 3 - Meet the staff of the Department

Develop a Cocol grammar that will describe a list of the names of staff who have optional titles, first names and/or initials, surnames, and (usually) qualifications, for example (`Staff.txt`):

Professor Shaun Bangay, PhD.
Professor P. D. Terry, MSc, PhD.
George Clifford Wells, BSc(Hons), MSc, PhD.
Greg G. Foster, PhD.
Dave A. Sewry, PhD.
Dr Karen Lee Bradshaw, MSc, PhD.
Mrs Madeleine Wright, MA, MSc.
C. Hubert H. Parry, BMus.
Dr Dameon Wagner, PhD.
Ms Busi Mzangwa.

Hint: notice the rather critical placing of commas and periods in this file. The exercise is partly one of being able to define tokens sensibly. Try to do this realistically so far as names are concerned, and limit yourself to a few examples of qualifications only.

Task 4 - So what if Parva is so restrictive - fix it!

Parva really is a horrid little language, isn't it? But its simplicity means that it is easy to devise Terry Torture on the lines of "extend it".

In the prac kit you will find the grammar for the first level of Parva, taken from page 164. Generate a program from this that will recognise or reject simple Parva programs, and verify that the program behaves correctly with two of the sample programs in the kit, namely `VOTER.PAV` and `VOTER.BAD`.

```
cmake Parva
crun Parva voter.pav
crun Parva voter.bad  -L
```

Now modify the grammar to add various features. Specifically, add (and check that the additions work):

- The `%` operator
- A *repeat-until* loop, and *break* and *continue* statements
- Increment and decrement statements like `Curse++;` `--Temper;` and `Bug[N]--;` (treat these as statements, not as components of expressions).
- A *for* loop inspired by the one in Pascal and as an alternative, one inspired by the one in Python.
- A restriction that a number cannot start with the digit 0 (unless this is the only digit).
- An expression operator hierarchy that matches the one in C/Java rather than the one in Pascal/Modula.

- An optional *else* clause for the *if* statement.
- An *in* operator that can test for membership of a list of values.
- A character type, and functions for converting from character values to integer values and *vice versa*.

Here are some silly examples of code that should give you some ideas:

```
void main () {
// (not supposed to do anything useful!)
  int i = 1, k = 0;
  while (i < 10) {
    do {
      k = k + 1;
      if (k > 5) break;
    } while (k < i);
    i = i + 1;
  }
  repeat
    read(k);
    write(i, i * k);
  until (k == 100);
  for i = 1 to 11 by 2 write(i);
  for i in ( 3, 5, 7, k, 2 * k ) write(i);
} // main


void main () {
// (not supposed to do anything useful!)
  int i = 0, j = 8;
  char ch;
  for ch = 'z' downto 'a' {
    write(ch);;;;
    write(char(ch + 4), int(ch), i + j * ch);
  }
} // main


void main () {
// (not supposed to do anything useful!)
  int age;
  bool beenKissed;
  read("How old are you, and have you been kissed? ", age, beenKissed);
  if (age == 16) {
    write("sweet sixteen");
    if (! beenKissed) write(" and never been kissed");
  }
  else
    if (age == 21) {
      write("party time!");
      int headache = 0, strain = 0;
      for beers = 20 to 0 by -1 {
        strain++; ++headache;
        if (strain % 8 == 0) {
          write("That\'s better"); strain = 0;
        }
      }
    }
  else if (age > 21 && age < 40) write("over the hill, bru");
  else if (age > 70)
    write("take a new lover");
  else
    write("boring");
} // main
```

These little programs and some other like them are in the kit, and you can easily write some more of your own.

Note: Read that phrase again: "that should give you some ideas". And again. And again. Don't just rush in and write a grammar that will recognise only some restricted forms of statement. Think hard about what sorts of things you can see there, and think hard about how you could make your grammar fairly general.

Hint: All we require at this stage is the ability to *describe* these features. You do *not* have to try to give them any semantic meaning or write code to allow you to use them in any way. In later pracs we might try to do that, but please stick to what is asked for this time, and don't go being over ambitious.

Task 5 - When all else fails, look up what you need in the index

In the prac kit you will find an extract from the index to my forthcoming bestseller "Hacking out a Degree", in the file `Index.txt`. It looks something like this:

```
abstract class 12, 45
abstraction, data 165
advantages of Java and Modula-2 1-99, 100-500, Appendix 4
aegrotat examinations -- see unethical doctors
aggregate pass, chances of 0
class attendance, intolerable 15, 745
class members 31
deadlines, compiler course -- see sunrise
horrible design (C and C++) 34, 45, 56-80
lectures, missed 1, 3, 5-9, 12, 14-17, 21-25, 28
loss of DP certificate 2010
no chance of supp 2010
prac tests 10, 25, 27, 30
probable exclusion from Rhodes 2011
recursion -- see recursion
senility, onset of 21-24, 105
subminimum 40
supplementary exams (first courses only) 45 - 49
wasted years 2007 - 2010
```

Develop a grammar `Index.atg` that describes this and similar indexes (indices?) and create a program that will analyse an index and accept or reject it (syntactically).

Appendix: Practical considerations when using Coco/R

For ease of use with the Java file and directory naming conventions, please

- **Do not use folder names (directory names) with spaces in them, such as "Prac 21"**
- Use a *fairly short* name (say 5 characters) for your goal symbol (for example, `Gram`);
- Remember that this name must appear after `COMPILER` and after `END` in the grammar itself;
- Store the grammar in a file with the same short primary name and the extension `.atg` (for example `Gram.ATG`).
- If required, store ancillary source code files in the subdirectory named `Gram` beneath your working directory. (Nothing like this should be needed this week.)

Make sure that the grammar includes the "pragma" `$CN`. The `COMPILER` line of your grammar description should thus always read something like

```
COMPILER Gram $CN
```

The laboratory installation of UltraEdit has been configured to link to Coco/R by using an option in the "advanced" pull down menu. To apply Coco/R to the file in the "current window", invoke UltraEdit from your working directory, for example

```
UEDIT32 Gram.atg
```

It is easy enough to configure a copy of UltraEdit you might have installed on your own computer to incorporate the option to invoke Coco/R. Use the `Advanced ->; Tool Configuration` pull down, and set up an option to read as in the example below

- The Command line should read **cmake %n**
- The Path line should read **%p**
- The Menu Item Name should be **Coco**
- Select the "Output to list box" option
- Tick the "Capture Output" option
- Remember to **Insert** the command into the menu

If the Coco/R generation process succeeds, the Java compiler is invoked automatically to try to compile the

application. If this does not succeed, a Java compiler error listing is redirected to the file ERRORS, where it can be viewed easily by opening the file in UltraEdit.

Free standing use of Coco/R

You can run the Java version of Coco/R in free standing mode with a command like:

```
cmake Gram
```

Like that, error messages are a little cryptic. In the form

```
cmake Gram -options m
```

the system will produce you a listing of the grammar file and the associated error messages, if any, in the file LISTING.TXT. If the Coco/R generation process succeeds the Java compiler is invoked automatically to try to compile the application. If this does not succeed, a Java compiler error listing is redirected to the file ERRORS, where it can be viewed easily by opening the file in UltraEdit.

Error checking

Error checking by Coco/R takes place in various stages. The first of these relates to simple syntactic errors - like leaving off a period at the end of a production. These are usually easily fixed. The second stage consists of ensuring that all non-terminals have been defined with right hand sides, that all non-terminals are "reachable", that there are no cyclic productions, no useless productions, and in particular that the productions satisfy what are known as **LL(1) constraints**. We shall discuss LL(1) constraints in class in the next week, and so for this practical we shall simply hope that they do not become tiresome. The most common way of violating the LL(1) constraints is to have alternatives for a nonterminal that start with the same piece of string. This means that a so-called LL(1) parser (which is what Coco/R generates for you) cannot easily decide which alternative to take - and in fact will run the risk of going badly astray. Here is an example of a rule that violates the LL(1) constraints:

```
assignment =   variableName ":=" expression
             | variableName index ":=" expression.
index      =   "[" subscript "]" .
```

Both alternatives for assignment start with a variableName. However, we can easily write production rules that do not have this problem:

```
assignment =   variableName [ index ] ":=" expression .
index      =   "[" subscript "]" .
```

A moment's thought will show that the various expression grammars that we have discussed in class - the left recursive rules like

```
expression = term | expression "-" term .
```

also violate the LL(1) constraints, and so have to be recast as

```
expression = term { "-" term } .
```

to get around the problem.

For the moment, if you encounter LL(1) problems, please speak to the long suffering demonstrators, who will hopefully be able to help you resolve all (or most) of them.